

UNIVERSITATEA "ALEXANDRU-IOAN CUZA" DIN IASI

**FACULTATEA DE INFORMATICA**



LUCRARE DE LICENTA

**Generare Procedurală  
Folosind Modele Markov**

propusă de

**Alexandru Loghin**

**Sesiunea:** iulie, 2019

Coordonator științific

**Conf. Dr. Anca Vitcu**

UNIVERSITATEA "ALEXANDRU-IOAN CUZA" DIN IASI

**FACULTATEA DE INFORMATICA**

**Generare Procedurală  
Folosind Modele Markov**

**Alexandru Loghin**

**Sesiunea:** iulie, 2019

Coordonator științific

**Conf. Dr. Anca Vitcu**

Avizat,

Îndrumător lucrare de licență,

Conf. Dr. Anca Vîtcu.

Data: .....

Semnătura: .....

## **Declarație privind originalitatea conținutului lucrării de licență**

Subsemnatul **Loghin Alexandru** domiciliat în **România, jud. Iași, mun. Pașcani, Ale. 1 Decembrie 1918, nr. 1, bl. C17, sc.B et. 1, ap. 27**, născut la data de **26 decembrie 2019**, identificat prin CNP **1971226225902**, absolvent al Facultății de informatică, **Facultatea de informatică** specializarea **informatică**, promoția 2019, declar pe propria răspundere cunoscând consecințele falsului în declarații în sensul art. 326 din Noul Cod Penal și dispozițiile Legii Educației Naționale nr. 1/2011 art. 143 al. 4 și 5 referitoare la plagiat, că lucrarea de licență cu titlul **Generare Procedurală Folosind Modele Markov** elaborată sub îndrumarea domnului **Conf. Dr. Anca Vîtcu**, pe care urmează să o susțin în fața comisiei este originală, îmi aparține și îmi asum conținutul său în întregime.

De asemenea, declar că sunt de acord ca lucrarea mea de licență să fie verificată prin orice modalitate legală pentru confirmarea originalității, consumând inclusiv la introducerea conținutului ei într-o bază de date în acest scop.

Am luat la cunoștință despre faptul că este interzisă comercializarea de lucrări științifice în vederea facilitării falsificării de către cumpărător a calității de autor al unei lucrări de licență, de diplomă sau de disertație și în acest sens, declar pe proprie răspundere că lucrarea de față nu a fost copiată ci reprezintă rodul cercetării pe care am întreprins-o.

Data: .....

Semnătura: .....

## **Declarație de consimțământ**

Prin prezenta declar că sunt de acord ca lucrarea de licență cu titlul **Generare Procedurală Folosind Modele Markov**, codul sursă al programelor și celealte conținuturi (grafice, multimedia, date de test, etc.) care însătesc această lucrare să fie utilizate în cadrul Facultății de informatică.

De asemenea, sunt de acord ca Facultatea de informatică de la Universitatea "Alexandru-Ioan Cuza" din Iași, să utilizeze, modifice, reproducă și să distribuie în scopuri necomerciale programele-calculator, format executabil și sursă, realizate de mine în cadrul prezentei lucrări de licență.

Absolvent **Alexandru Loghin**

Data: .....

Semnătura: .....

# Cuprins

<b>Motivăție</b>	<b>1</b>
<b>Introducere</b>	<b>3</b>
<b>1 Aspecte teoretice</b>	<b>4</b>
1.1 Preambul . . . . .	4
1.2 Modele Markov cu stări invizibile . . . . .	5
1.3 Estimare de parametri . . . . .	7
1.4 Algoritmi pentru estimare de parametri . . . . .	8
1.5 Limitări ale modelului . . . . .	11
1.6 Complexitate și metode de optimizare . . . . .	12
<b>2 Tehnologii</b>	<b>13</b>
2.1 Structura proiectului . . . . .	14
2.2 Componente . . . . .	15
2.2.1 MenuController . . . . .	16
2.2.2 GameController . . . . .	17
2.2.3 PlatformBuilder . . . . .	18
2.2.4 PlatformController . . . . .	18
<b>3 Arhitectura aplicației</b>	<b>19</b>
3.1 Detalii legate de implementarea librăriei . . . . .	20
3.2 Detalii despre sistemul de instantiere . . . . .	23
3.3 Utilizarea librăriei în joc . . . . .	24
3.4 Detalii legate de aspectul aplicației . . . . .	26
3.5 Detalii legate de controlul mașinilor . . . . .	30
3.6 Structura întregii aplicații . . . . .	31

<b>Concluzii</b>	<b>33</b>
<b>Bibliografie</b>	<b>34</b>
3.7    Cărți și Articole . . . . .	34
3.8    Obiecte preluate și folosite în cadrul aplicației . . . . .	35

# Motivație

Datorită pasiunii mele pentru *gamedev* și algoritmică am decis să realizez o modalitate de a genera procedural un mediu căt mai diversificat. Folosirea modelelor Markov a venit din necesitatea de a scăpa cumva de metoda clasică prin care se generează un șir de obiecte într-un mod aleator iar apoi se încearcă validarea acesteia după anumite constrângeri. Folosind acest model stocastic putem controla apariția unei subsecvențe prin stabilirea unei anumite probabilități de a se produce așadar eliminând cu totul pasul de validare.

Am decis așadar să încerc o abordare nouă, folosindu-mă de un model Markov ce a fost antrenat să respecte anumite reguli pentru a genera spațiul 3D. Scopul acestei lucrări este de a aduce o nouă perspectivă asupra generării procedurale și de a demonstra validitatea acestei abordări.

Pentru a arăta capabilitățile modelului, în cadrul proiectului a fost dezvoltată o librărie ce ușurează procesul de generare. De asemenea a fost nevoie și de un sistem ce facilitează plasarea obiectelor în mediul 3D, modelul Markov fiind folosit doar ca și generator. Odată funcționale mi-am îndreptat atenția asupra interfeței grafice și a interacțiunii dintre jucător și joc.

Datorită motorului grafic folosit și a instrumentelor de care dispune, timpul necesar pentru crearea jocului a fost îmbunătățit substanțial. De asemenea cu prilejul acestei aplicații am avut posibilitatea de a profunda și testa diferite tehnici de redare și post-procesare. Câteva dintre aceste elemente includ SSR<sup>1</sup>, lumină volumetrică, reflexii în timp real, corectare de culoare și multe altele.

Din punct de vedere al generării procedurale folosind modele Markov există articole ce prezintă folosirea acestora însă nu am găsit nici o librărie pentru un motor de joc ce implementează această abordare într-un mod facil și eficient. Ba chiar mai mult de obicei orice încercare descrisă în aceste publicații omit complet partea de învățare

---

<sup>1</sup>Screen Space Reflection

automată a acestor modele, ceea ce m-a determinat să abordez această temă.

În mod normal generarea procedurală folosește în spate un algoritm precum *Diamond – square*, sau un generator de zgomot precum *Perlin noise*. Abordarea folosind modelele Markov este mai facilă doarece pot fi foarte ușor antrenate să respecte un anumit set de constrângeri în mod dinamic, permitând o mai mare expresivitate și putere de modelare.

Desigur ca și celelalte metode acesta prezintă anumite dezavantaje, deși minore, pot influența decizia de a folosi acest tip de generare. Librăria creată în cadrul acestui proiect are ca scop promovarea acestui model căt și validarea lui, prin ușurința de utilizare căt și flexibilitatea oferită.

Unul din aceste dezavantaje este dezvoltare sa în *Unity* doarece are anumite dependințe de structura acestuia, dar ea poate fi ușor portată pe altă platformă întrucât în cadrul secțiunii teoretice este prezentat în detaliu un pseudocod ce descrie modul de funcționare. De asemenea algoritmul ce se ocupă cu antrenare acestor modele este foarte ușor de înțeles și folosit, necesitând doar o librărie de calcul numeric ce lucrează cu matrici.

De asemenea un obiectiv în crearea acestui pachet ce modelează un model Markov cu stări invizibile a fost eficiență, tocmai pentru a putea fi folosit într-o gamă mai largă de aplicații, incluzând cele de tip mobile. Cu toate acestea aplicația este solicitantă grafic din cauza tehnicilor de îmbunătățire a imaginii, aşadar un port pentru dispozitivele portabile este exclus, fiind posibilă doar dacă se reduce din complexitatea vizuală.

Odată cu finalizarea acestei aplicații, din cauza naturii sale *open source*, se va putea folosi librăria creată în cadrul acesteia pentru orice tip de proiect fără constrângeri, fiind ușor adaptabilă pentru orice sarcină ce necesită generare procedurală, de la muzică până la construcția unui mediu virtual, librăria fiind ușor de folosit și axata pe eficiență.

# Introducere

Având în vedere necesitatea creării unor medii căt mai versatile ce respectă anumite condiții sau restricții fizice s-au dezvoltat anumite tehnici pentru a facilita construcția automată și randomizată a oricărui element ce intră în componentă unui joc , de la modele tridimensionale până la coloana sonoră.

Tema lucrării s-a ivit din cauza necesității de adaptare rapidă a condițiilor folosite pentru generarea obiectelor din componența unui joc. Unul din avantajele acestei abordări este flexibilitatea oferită de modelul Markov fiind capabil să adapteze constrângerile folosite la generare cu o simplă reantrenare a modelului.

Aplicația prezentă este construită folosind un motor grafic pe care am dezvoltat o librărie capabilă să reprezinte un model Markov și să îl antreneze. Folosindu-mă de această librărie , pot genera o secvență de obiecte ce încearcă să respecte fidel constrângerile folosite la antrenare.

Procesul de generare procedurală este compus din două etape. Generarea obiectelor și plasarea lor convenabil în spațiul virtual. Am încercat în special să decuplez cele două etape pentru o mai bună modularizare și o coeziune ridicată. Cele două sisteme lucrează independent unul față de celălalt , relația dintre cele două fiind una de agregare.

Cele două sisteme sunt puse în funcțiune pentru a genera în timp real mediul pentru jucător, mediu ce este împărțit în trei zone de interes , urbana , rurală și desertică. De asemenea pentru a adauga complexitate se alternează între trei tipuri de platforme ce simulează un drum drept, un drum cu viraj la stânga sau un drum cu viraj la dreapta.

# Capitolul 1

## Aspecte teoretice

### 1.1 Preambul

În domeniul probabilităților un model Markov este folosit pentru a modela un sistem ce se schimbă într-un mod aleator. De obicei acesta ține cont doar de starea curentă și nu depinde de evenimentele aleatoare, acest lucru numindu-se și proprietatea Markov. Ulterior s-au dezvoltat modele ce au un astfel numit ordin, extinzând astfel numărul de stări de care se ține cont în cadrul modelului.

De obicei există o separație clară între tipurile de modele Markov, criteriul folosit este disponibilitatea de a observa sau nu stările modelului. Așadar rezultă două mari categorii, sisteme cu stări complet observabile, din care fac parte lanțurile Markov și sisteme cu stări parțial observabile, cel mai cunoscut sistem fiind modelul Markov cu stări invizibile sau HMM<sup>1</sup>. Pe langă aceste sisteme prezentate, s-au dezvoltat și anumite derivate fiecare cu avantajele sale demonstrând astfel flexibilitatea și capacitatea de modelare a acestor sisteme.

În mare parte această teză se axează în jurul partii discrete a acestor modele, numărul de stări/observații fiind finit numărabil și ușor de caracterizat dar există și o ramură ce se ocupă cu partea continuă, fiecare abordare având avantajele și dezavantajele ei.

Legat de partea practică, de-a lungul timpului aceste sisteme au fost folosite în foarte multe domenii de la bioinformatică până la lingvistică computațională. O aplicație foarte importantă a acestor modele este recunoașterea vocală, modelele Markov fiind standardul folosit în industrie pentru asistenții personali ca *Siri* și *Alexa*.

---

<sup>1</sup>Hidden Markov Model

## 1.2 Modele Markov cu stări invizibile

Acest model statistic este caracterizat în principal de faptul că stările interne sunt ascunse de un privitor exterior. Tot ce se poate observa în cadrul acestui model este emisia unor etichete sau obiecte  $\{v_1, v_2, v_3, \dots, v_n\}$  dintr-o mulțime notată pe scurt  $V$ . Acest lucru complica în general structura modelului și algoritmii ce folosesc acest sistem.

Un avantaj direct este creșterea capacitatii de expresivitate, putând modela mai fidel datele, datorită eliminării necesității de a cunoaște toate stările evenimentului.

Pentru a caracteriza concret și complet acest model avem nevoie de un *5-uplu HMM* = ( $\mathbf{S}, \mathbf{V}, \mathbf{A}, \mathbf{B}, \pi$ ) ce desemnează astfel :

- $\mathbf{S} = \{S_1, S_2, \dots, S_N\}$  mulțimea ce desemnează numărul de stări ascunse din model, avand un numar de  $N$  elemente. Starea relativa la timpul  $t$  se va nota ca și  $q_t$ .
- $\mathbf{V} = \{V_1, V_2, \dots, V_M\}$ , cele  $M$  etichete observabile.
- $\mathbf{A} = \{a_{ij}\}$  unde  $a_{ij} = P(q_{t+1} = S_j | q_t = S_i), 1 \leq i, j \leq N$ , reprezentând distribuția de probabilitatea asociată tranzițiilor între stări.
- $\mathbf{B} = \{b_i(k)\}$  unde  $b_i(k) = P(V_k la timpul t | q_t = S_i), 1 \leq i \leq N, 1 \leq k \leq M$ , ce caracterizează distribuția de probabilitate asociată emiterii unui element din  $V$  în starea  $i$ .
- $\pi = \{\pi_i\}$  unde  $\pi_i = P(q_1 = S_i), 1 \leq i \leq N$ , folosit inițial pentru a stabili prima stare din model și anume  $q_1$ .

În general modul de funcționare a acestui model poate fi descris foarte ușor cu o bucătă de pseudocod:

Exemplu 1.1: Pseudocod ce descrie modul de operare a unui *HMM*

```
1   t ← 1;  
2   stare ← alege o stare  $S_i$  cu probabilitate  $\pi_i$ ;  
3   repeta la infinit  
4       stare ← alege o nouă stare  $S_j$  de tranziție de la starea  $S_i$  cu probabilitate  $a_{ij} \in \mathbf{A}$ ;  
5       afiseaza observatia  $V_k$  cu probabilitatea  $b_i(k) \in \mathbf{B}$ ;  
6       t ← t + 1;
```

Uneori printre alte publicații de specialitate modelul poate fi descris prin specificarea parametrilor  $N, M$  și cele trei distribuții de probabilitate notate astfel  $\lambda = \{\mathbf{A}, \mathbf{B}, \pi\}$ .

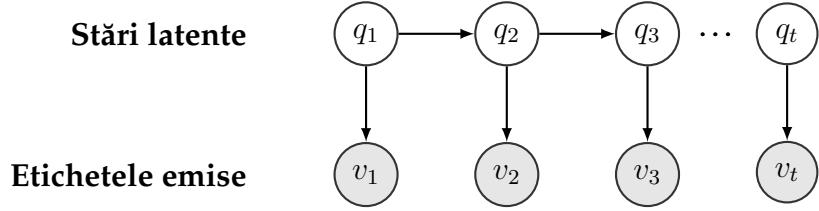


Figura 1.1: Modul de funcționare a modelului Markov cu stări invizibile

În cadrul aplicației multimea de etichete **V** au fost reprezentată de elementele de tip *GameObject* ce urmează a fi instanțiate. Din motive de performanță și flexibilitate aplicația folosește în total un număr de săse modele Markov cu stări invizibile a căror parametri au fost estimați să respecte anumite restricții.

Se poate face o distincție între cei săse generatori după scopul lor în cadrul jocului, trei dintre ei sunt folosiți pentru a genera terenul pe care jucătorul navighează. Această decizie a fost făcută din necesitatea de a putea controla numărul de platforme pentru fiecare din cei trei generatori, ce în fond reprezintă trei zone distincte, urbană, rurală și desertică enumerate conform ordinii de apariție în joc.

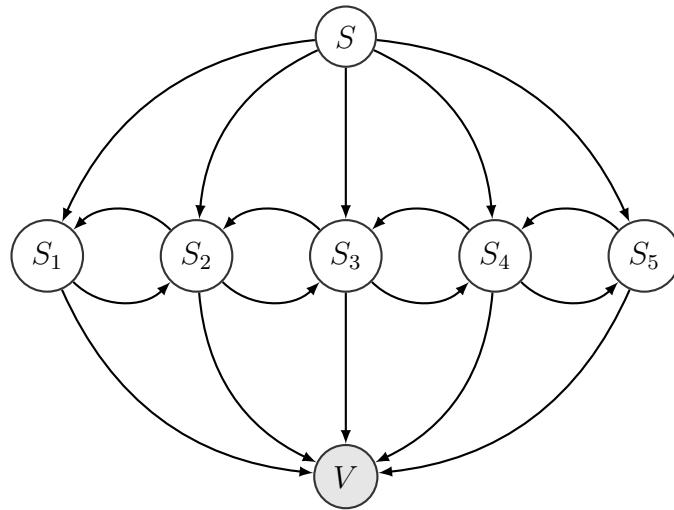


Figura 1.2: Structura unui generator din cele trei zone

Se observă din figură, că avem un nod de start **S** ce ne conduce la una dintre cele cinci stări, iar fiecare stare are acces la **V**, multimea etichetelor, în cazul nostru la cele trei tipuri de platformă *Left*, *Right* și *Straight*. Din model s-au omis probabilitățile de pe arce tocmai pentru a nu încărca desenul că și pentru faptul că ele vor fi inferate ulterior de un algoritm de estimare a parametrilor, inițial probabilitățile fiind necunoscute.

Restul de generatori sunt folosiți pentru obiectele ce urmează a popula platforme. Din cauza complexității și a numărului mare de stări și etichete nu se poate realiza o reprezentare grafică. Cel mai mare dintre generatori are douăzeci de stări și zece obiecte ce au rol de etichete pentru emisie.

O observație importantă legată de mulțimea  $V$  este că aceasta conține un element mai special și anume un *GameObject* denumit sugestiv epsilon. Acest obiect semnifică elementul vid, prin care se poate insera un spațiu între obiectele din scenă pentru un aspect mai placut la instanțierea pe platforme. Această decizie s-a luat din necesitatea de a separa unele obiecte de celelalte pentru a putea reproduce mai fidel lumea reală.

### 1.3 Estimare de parametri

Această dilema este una dintre cele trei mari aspecte a modelelor Markov cu stări invizibile, celelalte două fiind determinarea probabilității de emisie a unei secvențe și determinarea celei mai bune secvențe de stări ce descriu seria de emisii observate. Problema estimării de parametri este cea mai grea dintre cele trei, nici până în momentul de față nu s-a dezvoltat un algoritm simplu ce rezolvă această problemă. Majoritatea algoritmilor folosesc o metodă iterativă pentru a încerca o estimare a parametrilor ce descriu un HMM și anume  $\lambda = \{\mathbf{A}, \mathbf{B}, \pi\}$ .

Din cauză naturii iterative în general acești algoritmi se bazează pe convergența parametrilor pentru terminare. Un lucru bun este că s-a demonstrat apropierea de un anumit punct a acestor algoritmi ceea ce înseamnă ca se termină întotdeauna, cu toate acestea abordarea iterativă suferă de o convergență foarte înceată și de pericolul continuu de a rămâne blocat într-un optim local.

De aceea s-au încercat multe artificii pentru a obține un *log – likelihood* căt mai bun. Unul dintre ele, folosit și de această aplicație, este inițializarea aleatoare a parametrilor  $\lambda = \{\mathbf{A}, \mathbf{B}, \pi\}$  și încercarea antrenării până la convergență de mai multe ori pastrându-se cel mai bun rezultat. De asemenea numărul de stări latente poate influența drastic comportamentul modelului, fiind un factor decisiv între *underfit* și *overfit*.

Din păcate determinarea acestui număr de stări optim pentru un model dat este o problemă grea ce nu are o soluție fixă. În general abordarea acestei probleme este prin *trial and error*. Există și anumite criterii bazate pe *log – likelihood* și numărul

parametrilor din model, cum ar fi  $BIC$ <sup>2</sup> sau  $AIC$ <sup>3</sup>.

În cadrul acestei proiect a fost folosită metoda *trial and error* în defavoarea criteriilor prezentate mai sus doarece, modelele sunt relativ mici și nu necesită cei mai optimi parametri. În general s-a urmărit ca numărul de stări să fie mai mare decât numărul de etichete, ceea ce previne de fapt inabilitatea rețelei de a învăța restricțiile la antrenare. Acest lucru se mai numește și *underfit*.

## 1.4 Algoritmi pentru estimare de parametri

Cel mai cunoscut algoritm pentru inferare de parametri, fiind dat un model Markov cu stări invizibile este aşa numitul algoritm *Baum–Welch* sau algoritmul *Forward–Backward*. Aşa cum implică și numele acest algoritm are la bază două etape. Inițial parametri modelului  $\lambda = \{\mathbf{A}, \mathbf{B}, \pi\}$  sunt inițializați aleator și mai apoi actualizați după fiecare iterație până la convergență după cum urmează:

- **Forward** este etapa în care se calculează de obicei  $P(O|\lambda)$ , unde  $O$  este secvența de etichete observate, folosind tehnica programării dinamice pentru stocarea anterioră a rezultatelor. Acest calcul este necesar doarece va fi folosit în urmatoarea etapă pentru reajustarea parametrilor modelului.
- **Backward** este etapa de calcul a probabilității  $P(O_{t+1}, \dots, O_T | S_i, \lambda)$ , unde  $T$  este lungimea de etichete observate, ceea ce semnifică șansa de a observa secvența  $O_{t+1}, \dots, O_T$  aflându-ne în starea  $i$  și la momentul  $t$  din timp.

În ciuda popularității acestui algoritm am decis să folosesc altă modalitate de estimare a parametrilor din motive de viteză și simplitate atât la nivel de implementare căt și la nivel teoretic. Publicația ce descrie acest algoritm prezintă grafice convingătoare în privința vitezei atunci când lungimea secvenței observate  $O$  depășeste numărul de etichete posibile al modelului.

Un alt aspect alt acestei abordări este că algoritmul actualizează doar parametri  $\mathbf{A}, \mathbf{B}$  ai modelului ignorând complet parametrul  $\pi$  ceea ce duce la o amplificare a vitezei cu consecințe minime, parametrul  $\pi$  fiind folosit doar la alegerea stării initiale  $q_1$ .

De asemenea ca și *Baum – Welch*, fiind dată o secvență  $O$  de etichete observate de lungime  $T$ , antrenare nu se poate face pe o subsecvență  $O_1, \dots, O_m$  iar mai apoi pe

---

<sup>2</sup>Bayesian Information Criterion

<sup>3</sup>Akaike Information Criterion

o alta subsecvență  $O_{m+1}, \dots, O_T$ , cu  $1 \leq m \leq T$ , deoarece modelul ar avea parametri adaptați doar după ultima subsecvență pe care a fost antrenat. Un alt aspect important este normalizarea valorilor ce se face la fiecare iterație după actualizarea parametru- lor. Acest lucru este necesar doarece fiecare tranziție și emisie depinde în fond de o variabilă aleatoare a carei sumă de probabilități trebuie să fie exact egală cu unu.

Algoritmul pe care am decis să îl folosesc are la bază o matrice  $C$  de dimensiune  $M \times M$  construită cu scopul de a caracteriza numărul de apariții a unei anumite perechi  $O_t O_{t+1}$  în secvența de etichete observate  $O$ .

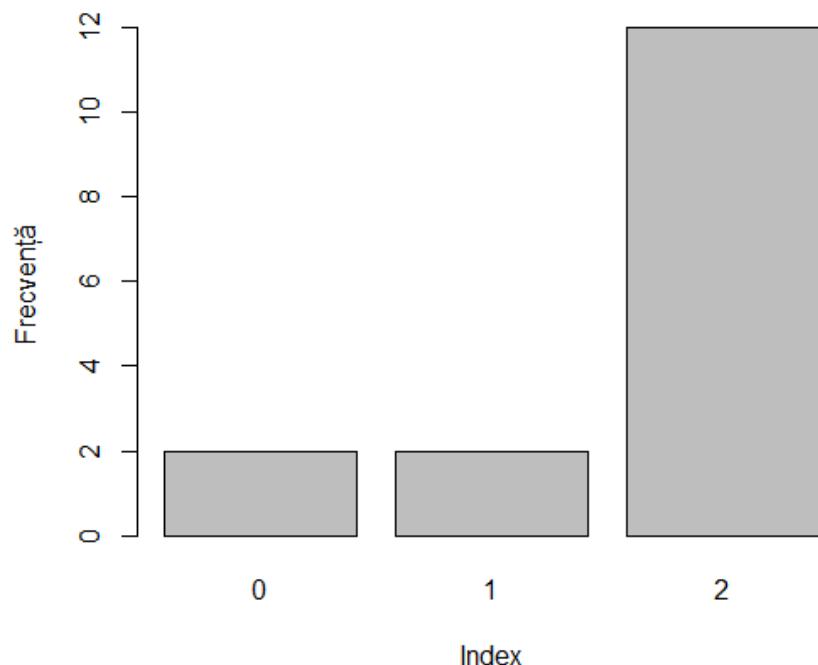


Figura 1.3: Histogramă a unei secvențe de etichete folosite la antrenare

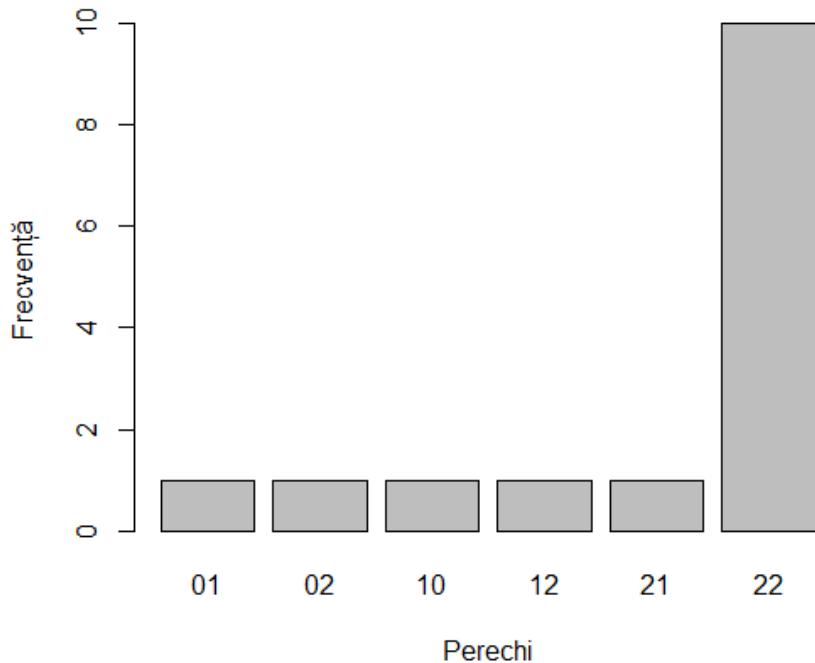


Figura 1.4: Histogramă a perechilor de etichete din secvența folosită la antrenare

Această soluție se bazează foarte mult pe calculul matriceal de accea în cadrul aplicației am fost nevoit să apelez la o librărie ce este folosită pentru calcul numeric. Deoarece acest algoritm se bazează foarte mult pe matrici, o încercare de implementare folosind unități de procesare grafice ar putea îmbunătăți substanțial viteza de convergență a algoritmului datorită puterii mai mari de calcul.

#### Exemplu 1.2: Pseudocod ce descrie algoritmul de inferență

```

1 Intrare: Matricea C, A, BT
2 repeta pana la convergenta:
3   R ← C ⊙ C
4   A' ← A ⊙ (BTRB)
5   B' ← B ⊙ (RBAT + RTBTA)
6   A ← norm(A')
7   B ← norm(B')
8   A ← norm(A')

```

Parametri  $\mathbf{A}$ ,  $\mathbf{B}$ ,  $\pi$  sunt initializați aleator și normalizați pe linii, iar apoi este calculat  $\bar{\mathbf{C}} = \mathbf{B}\bar{\mathbf{A}}\mathbf{B}^T$ , unde  $\bar{\mathbf{A}} = \pi_k \cdot a_{kl}$ ,  $1 \leq k \leq N$ ,  $1 \leq l \leq N$ . Următorul pas este calcularea matricii  $\mathbf{R}$  prin împărțirea element cu element a matricii  $\mathbf{C}$  la matricea  $\bar{\mathbf{C}}$ , notat cu  $\odot$  în cadrul pseudocodului.

După calculul acestor matrici intermedii urmează actualizare parametrilor după urmatoarele ecuații,  $\bar{\mathbf{A}}' = \bar{\mathbf{A}} \odot \mathbf{B}^T \mathbf{R} \mathbf{B}$  și  $\mathbf{B}' = \mathbf{B} \odot (\mathbf{R} \bar{\mathbf{B}} \bar{\mathbf{A}}^T + \mathbf{R}^T \mathbf{B} \bar{\mathbf{A}})$ , unde  $\odot$  repre-

zintă înmulțirea element cu element a două matrici. După acest calcul se face actualizarea astfel  $\mathbf{A} = \text{norm}(\bar{\mathbf{A}}')$ ,  $\mathbf{B} = \text{norm}(\mathbf{B}')$ , unde  $\text{norm}$  este o funcție ce realizează normalizarea pe toată matricea  $\mathbf{A}$  iar pentru  $\mathbf{B}$  doar pe coloane. De asemenea se realizează și o normalizare a matricii  $\mathbf{A}$  pe linii, la finalul iterățiilor.

Acești pași se realizează până un anumit set de criterii de convergență sunt satisfăcute. De obicei aceste criterii includ ca norma matricilor  $\mathbf{A}, \mathbf{B}$  să fie sub un anumit epsilon împreună cu funcția de *log – likelihood*. În implementarea acestui algoritm am folosit ca și criterii de convergență, cum sugera și publicația de unde am preluat ideea, calcularea unei norme de tip  $L^2$  aplicată matricilor  $\mathbf{A}, \mathbf{B}, \mathbf{R}$ , ce ar trebui să fie sub un  $\epsilon = 10^{-6}$  și un număr maxim de iterări pentru care ar trebui să fie îndeplinită. Bineînțeles aceste condiții sunt doar cele standard, ele putând fi ușor schimbate în funcție de necesitate.

## 1.5 Limitări ale modelului

În secțiunile anterioare am văzut avantajele modelului Markov cu stări invizibile dar acesta suferă de blocarea într-un optim local. Din cauza acestei problemele estimarea de parametri este o sarcină destul de dificilă și delicată. De asemenea modelul depinde foarte mult de proprietatea Markov ce specifică clar că o stare viitoare poate să depindă doar de starea curentă, limitând astfel capacitatea de a modela date mult mai complexe. Se poate extinde modelul la ultimele  $n - 1$  stări dar complexitatea de antrenare și durata crește considerabil, se mai numește și model cu memorie.

Cea mai simplă soluție pentru a preveni blocarea este inițializarea aleatoare a parametrilor modelului și reantrenarea lui de mai multe ori, pastrandu-se cel mai bun rezultat. Această soluție este fezabilă deoarece prin randomizare se pot obține parametri din vecinătatea soluției optime, scăzând posibilitatea ca algoritmul să ramană blocat într-un optim local, atunci când există o multitudine de aceste puncte. Desi această soluție rezolvă problema în unele situații reantrenarea de foarte multe ori este costisitoare, de accea există posibilitatea de a salva parametri estimati.

Din cauza acestor limitări algoritmii de estimare a parametrilor precum *Baum – Welch* nu garantează găsirea optimului global. Deseori un optim local este suficient pentru probleme ce nu necesită soluția optimă dar trebuie luat în considerare acest aspect atunci când se utilizează un astfel de algoritm.

O altă problemă a acestei abordări este că în general este asumată independența

etichetelor una față de celălaltă limitând astfel flexibilitatea modelului și capacitatea de modelare a datelor.

## 1.6 Complexitate și metode de optimizare

În această secțiune se vor prezenta probleme legate de complexitatea algoritmului de antrenare căt și a algoritmului de eșantionare dintr-o distribuție de probabilitate discretă. Algoritmul de estimare a parametrilor<sup>[1]</sup> prezentat anterior ce se folosește de o matrice de apariție  $\mathbf{C}$  are complexitatea timp de ordinul  $O(IM^2N + T)$ , unde  $I$  reprezintă numărul de iterații până la convergență.

Comparativ cu algoritmul *Baum – Welch* ce are ca și complexitate timp  $O(IN^2T)$  iar în practică secvența de etichete pe care e antrenat algoritmul are o lungime considerabil mai mare decât cardinalul mulțimii  $V$ , adică  $M$ , de unde se observă că abordarea cu matricea de apariție aduce un surplus de rapiditate de procesare, ce este important în domeniul jocurilor unde eficiența este un aspect cheie.

Legat de procesul de eșantionare, numit și *Inverse transform method*, acesta este realizat prin calcularea distribuției  $CDF$ <sup>4</sup> pentru parametri  $A, B$  pentru a putea realiza o căutare binară a stării către care se va face tranzitia fiind dat un număr generat pseudorandom. Din punct de vedere al complexității sunt necesari pași adiționali cum ar fi stocarea valorilor cumulative dar asta se face în etapa de inițializare având efect minimal asupra jocului. Ce se poate spune totuși de această abordare este că timpul de căutare a stării se transformă din  $O(N)$ , unde  $N$  reprezintă numărul de stări, în  $O(\log(N))$ . Această optimizare este importantă deoarece operația de eșantionare are loc de zeci de ori pentru o platformă fiind practic cea mai folosită operație din cadrul librăriei.

Deși în practică metoda de *sampling* în timp de  $O(\log(N))$  s-a dovedit a fi suficientă, există un algoritm și mai eficient decât cel prezentat ce reușește să determine starea sau emisia find dat un număr generat aleator în timp de  $O(1)$ , având o complexitate la inițializare căt și memorie echivalentă cu cea a algoritmului de căutare binară.

Acest algoritm se numește *metoda lui Alias*, și reușește să obțina cea mai bună viteză posibilă pentru eșantionarea dintr-o distribuție de probabilitate discretă. În general ideea din spatele algoritmului este de a crea o altă distribuție pe baza celei de la intrare ce permite eșantionarea în timp constant.

---

<sup>4</sup>Cumulative Distribution Function

# Capitolul 2

## Tehnologii

În cadrul acestui proiect s-au folosit următoarele tehnologii :

- **Unity 2019.1b** : motorul grafic folosit pentru construirea și randarea jocului.
- **Blender** : program *open source* folosit pentru modelarea 3D și construcția obiectelor ce au fost ulterior importate în motorul grafic.
- **Math.NET** : librărie *open source* folosită pentru realizarea algoritmului de învățare automată deoarece acesta se bazează foarte mult pe lucru cu matrici.
- **C#** : limbajul nativ folosit de Unity pe lângă JavaScript , decizia fiind luată pur din motive de viteză.
- **Componente din Unity** : module *built – in* ce au ajutat la construirea aplicații de la coliziuni până la animații.

Din multitudinea de motoare grafice disponibile am ales **Unity** doarece e o tehnologie cu care am mai lucrat, fiind conștient de capabilitățile acestuia. Multitudinea de unelte îl fac foarte ușor de recomandat și împreună cu magazinul de *assets* m-au ajutat foarte mult în procesul de construcție al acestui proiect. De asemenea **Unity** dispune de o documentație foarte bogată și bine pusă la punct , calități cheie pentru aprofundarea și folosirea acestui motor grafic.

Câteva exemple de sisteme ce vin la pachet cu motorul grafic ar fi, motorul de simularea a fizicii, motorul audio, sistemul de prefabricate, sistemul de iluminare și randare etc. Fără folosirea unei tehnologii ca **Unity**, acest proiect nu ar fi fost posibil într-un timp atât de scurt , timpul de dezvoltare putând fi chiar triplat. Singurul dezavantaj imediat al acestui motor grafic este natura sa *closed source*.

Pentru modelele 3D folosite în cadrul jocului a fost necesară folosirea unei editor și anume **Blender**. Deși mare parte din obiectele importate în joc sunt preluate de pe magazinul integrat în **Unity**, există și anumite obiecte ce au fost modelate de către mine. Cu acest prilej am aprofundat anumite tehnici ce implică modelarea 3D , unul dintre cele mai importante fiind *UV unwrapping*.

Am ales **Blender** datorită politiei *open source* și a documentației vaste disponibile, fiind foarte ușor de lucrat în acesta, neputând aduce aplicația la un nivel dorit de finisaj fară acestă unealtă.

Din punct de vedere al librăriilor externe am folosit **Math.NET** , importată în **Unity** folosind **NuGet**. Necesitatea acestei librări este datorat algoritmului de invățare automată ce se bazează foarte mult pe lucrul și manipularea matricilor, de la adunarea cu un scalar până la înmulțirea/împărțirea element cu element. Detaliile legate de acest algoritm împreună cu avantajele și dezavantajele acestuia au fost discutate în secțiunea teoretică.

Datorită motorului grafic folosit și a limitărilor acestuia întregul proiect a fost construit folosind **C#**, alternativa sa fiind **JavaScript**. Din cauza experienței cu **C#** și a performanței ridicate am luat decizia de a folosi acest limbaj.

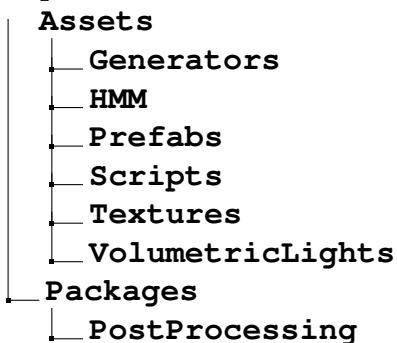
Legat de componentele folosite ce se găsesc în cadrul lui *Unity*, acestea au ajutat foarte mult la grăbirea procesului de creare a aplicației. Căteva dintre aceaste sunt **Inspector**, **WheelCollider**, **BoxCollider**, **RigidBody**, **Animator**, **Light** și multe altele.

## 2.1 Structura proiectului

Aplicația creată este împărțită în două etape , generare și instantiere. Din cauza unor motive de performanță fiecare zonă, desertică, rurală și urbană are propriul său generator. De asemenea fiecare tip de platformă folosește propriul său model Markov însumând astfel un total de șase generatori, ce se ocupă doar cu producerea unei secvențe specifice de obiecte. Pentru partea de instantiere am fost nevoit să construiesc un sistem ce facilitează instanțierea obiectelor generate, plasândule pe platformă în sensul acelor de ceasornic.

Legat de structura directoarelor, **Assets** conține toate elementele necesare aplicației, de la librăria folosită pentru generarea obiectelor până la modulele de sunet și interfață iar directorul **Packages** conține cele mai importante unele folosite pentru realizarea și finisajul jocului importate direct de către **Unity**.

Aplicația fiind facută în **Unity** fișierele și directoarele au o structură arborescentă după cum urmează:



**Generators** aici se află cei săse generatori folosiți pentru construirea mediului.

**HMM** este directorul ce conține scripturile necesare pentru construirea modelului lui Markov și a antrenării lui.

**Prefabs** este directorul cu toate prefabricatele din joc, incluzând modelele pentru platforme, obiecte, jucător cât și inamici.

**Scripts** aici sunt stocate toate fișierele sursă ce modeleză comportamentul jucătorului, al meniurilor și a sistemului de generare/instantiere, sumându-se în total la optsprezece fișiere.

**Textures** directorul ce conține toate fișierele de tip *Albedomap*, *Heightmap* și *Occlusionmap* folosite în cadrul materialelor ce au fost ulterior importate pe obiecte.

**VolumetricLights** directorul ce conține fișierele preluate de pe *Github* pentru lumină volumetrică.

**PostProcessing** modulul folosit de *Unity* pentru a putea integra efecte precum corectare de culoare, *bloom*, *motion blur* și multe altele.

## 2.2 Componente

Aplicația prezintă este împărțită pe diferite module ce controlează sunetul, generaarea, *UI-ul*, instantierea și multe altele. În **Unity** acest lucru se realizează în mare parte prin fișiere sursă atașate de obiectele din scena curentă.

Am optat să construiesc jocul în jurul unei singure scene, ceea ce necesită un numar mai mare de operatori și fișiere sursă. O consecință directă a folosirii unei singure scene este integrarea meniului principal în același mediu ca și jocul în sine. De aici derivă necesitatea de două scripturi ce controlează tranzitia de la meniu la joc intitulate sugestiv *MenuController* și *GameController*.

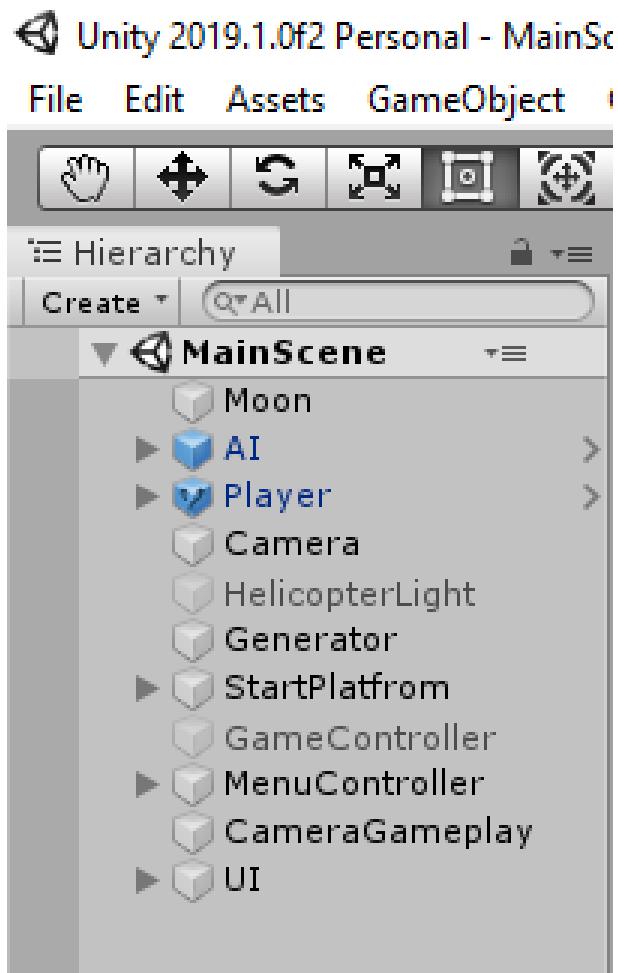


Figura 2.1: Structura arborescentă a obiectelor din scenă

De asemenea există un obiect *Generator* în scenă ce conține toți generatorii necesari pentru construirea mediului pentru jucător. Unul pentru fiecare tip de zonă de interes și tip de platformă. Legătura între cei săse generatori este facută de cele două scrip-turi atașate de obiectul *Camera* denumite *PlatformBuilder* și *PlatformController*. Primul se ocupă cu instantierea și plasarea obiectelor pe platformă în funcție de parametri de instanțiere setați de un script atașat de un *Spawn Point*. Cel de-al doilea se ocupă cu randomizarea și controlarea numărului de platforme pentru fiecare zonă de interes.

### 2.2.1 MenuController

Acest obiect se ocupă de toate aspectele ce țin de meniul principal. Deoarece există o singură scenă scriptul este destul de complex, rolul său fiind activarea tuturor componentelor ce țin de jocul principal, sunet, inamici, jucător, cameră, efecte și multe altele.

În continuare se vor prezenta cele mai importante secvențe de cod ce se găsesc în

fișierul **MenuController**.

Exemplu 2.1: Funcțiile din MenuController

```
1 public void OnGameExit(){
2     Application.Quit();
3 }
4 public static void OnGameReset(){
5     retry = true;
6 }
7 public void OnCredits(){
8     this.creditsStart = true;
9     this.credits.SetActive(true);
10}
11 public void OnGameStart(){
12     gameStart = true;
13     gameController.SetActive(true);
14     player.constraints = RigidbodyConstraints.None;
15     AI.constraints = RigidbodyConstraints.None;
16     player.velocity = Vector3.forward * 3;
17     AI.velocity = Vector3.forward;
18     bars.SetActive(true);
19 }
```

Cele patru funcții controlează tranzițiile aplicației în funcție de interacțiunea jucătorului cu meniul principal. De remarcat în funcția *OnGameStart* linia ce activează *GameController* odată ce se pornește jocul și anume *gameController.SetActive(true)*.

## 2.2.2 GameController

Obiectul ce se ocupă de funcționalitatea principală din joc. În mare parte meniul de pauză și meniul de final al jocului sunt controlate de acest script.

Legat de parte de cod, fișierul sursă atașat de obiect arată astfel:

Exemplu 2.2: Funcțiile din GameController

```
1 public void SetPausedState(){
2     gamePaused = gamePaused == true ? false : true;
3 }
4 public void SetGameOverState(){
5     player.GetComponent<CarController>().enabled = false;
6     cameraScript.enabled = false;
7     enemyScript.enabled = false;
8     gameOver = true;
9     gameOverScreen.SetActive(true);
10}
11 public void TogglePause(){
12     ToggleSound();
13     ToggleRender();
14     TogglePauseMenu();
15     SetPausedState();
16 }
```

Meniul din joc odată activat oprește actualizarea grafică a jocului, lucru ce complice apelul de funcții dependente de timp.

## 2.2.3 PlatformBuilder

Scriptul ce realizează construirea în timp real a platformelor și le instantiază. Datorită volumului mare de cod mai jos este prezentată doar funcția ce adaugă obiectele pe platformă. Înainte de instanțiere funcția are grija să preia datele de *spawn*, conținute de scriptul atașat *SpawnSettings* pentru fiecare loc valid de instanțiere.

Exemplu 2.3: Funcțiile din PlatformBuilder

```
1 public GameObject BuildPlatform(GameObject state){  
2     GameObject nextPlatform = Instantiate(state, new Vector3(currentPlatform.transform.position.x + xOffset, -15, currentPlatform.transform.position.z + zOffset),  
3         Quaternion.Euler(0, 0, 0));  
4     HMM propGenerator = ChoosePropGenerator(nextPlatform.name);  
5     Transform spawnPoints = nextPlatform.transform.Find("SpawnPoints");  
6     foreach (Transform spawn in spawnPoints){  
7         SpawnSettings spawnSettings = spawn.GetComponent<SpawnSettings>();  
8         if (spawnSettings.isStackable == false && spawnSettings.maxObjectStack > 1)  
9             throw new System.Exception("Spawn is not stackable!");  
10            for (int i = 0; i < spawnSettings.maxObjectStack; ++i){  
11                InstantiateProp(propGenerator, spawn, spawnSettings, i);  
12            }  
13            nextPlatform.transform.rotation = Quaternion.Euler(0, rotation, 0);  
14        return nextPlatform;  
15    }
```

Scriptul *SpawnSettings* conține doi parametri *isStackable* și *maxObjectStack* folosite pentru a specifica dacă locul de *spawn* poate să conțină mai multe obiecte și în ce cantitate.

## 2.2.4 PlatformController

Din cauza necesității de a controla durata unei anumite zone evitând astfel inconsistență, *PlatformController* alege un număr de platforme pentru fiecare secțiune ținând cont de o limită superioară și inferioară.

Exemplu 2.4: Funcțiile din PlatformController

```
1 public GameObject GetNextPlatform(){  
2     GameObject output;  
3     currentEmissionCount++;  
4     if (currentEmissionCount > maxEmissionCount){  
5         output = transitionPlatforms[currentGeneratorIndex];  
6         currentEmissionCount = 0;  
7         currentGeneratorIndex = (currentGeneratorIndex + 1) % generators.Length;  
8         maxEmissionCount = Random.Range(lowerBound, upperBound);  
9     }  
10    else{  
11        output = generators[currentGeneratorIndex].NextEmission();  
12    }  
13    return output;  
14 }
```

# Capitolul 3

## Arhitectura aplicației

În mare parte aplicația este construită în jurul unei singure scene folosind *Unity*, arhitectura aplicației depinzând în mare parte de uneltele disponibile în motorul grafic și modul în care acesta funcționează. Am încercat să împart căt de mult se poate aplicația în module cu o coeziune căt mai ridicată și cuplaj scăzut. De accea sistemul de instantiere, generare și amplasare a obiectelor este construit cu principalul scop de a fi modular. Chiar și librăria ce modelează prototipul Markov cu stări invizibile este decuplat de algoritmul de antrenare în sine, putând fi schimbat oricând fără a afecta funcționalitatea librăriei.

Aplicația folosește și multe funcții de *callback* pentru a determina când anumite elemente din cadrul jocului ar trebui activate. Spre exemplu cea mai folosită funcție este cea de *OnTriggerEnter*, ce determină când un *RigidBody*<sup>1</sup> se intersectează cu un *BoxCollider*<sup>2</sup>.

Exemplu 3.1: Exemplu de utilizare a functiei OnTriggerEnter

```
1 void OnTriggerEnter(Collider collidingObject){  
2     if (!this.triggered){  
3         this.callbackObject.GetComponent<PlatformBuilder>().InstantiatePlatform();  
4         this.triggered = true;  
5     }  
6 }
```

De asemenea *Unity* dispune de o unealtă foarte utilă numită *Inspector* ce permite asignarea de referințe a obiectelor din scenă în scripturi, lucru foarte util și eficient atunci când este necesară legarea anumitor module. De exemplu *callbackObject* reprezintă obiectul din scenă de care este atașat codul sursă ce se ocupă de construirea platformelor, numit sugestiv *PlatformBuilder*, din care se apelează funcția *InstantiatePlatform*

<sup>1</sup>Componenta ce determină dacă obiectul este supus motorului de fizica

<sup>2</sup>Componenta ce determină zona de coliziune a unui obiect

destinată acestui scop.

### 3.1 Detalii legate de implementarea librăriei

Cum menționam și anterior pentru implementare am avut nevoie de o librărie ce se ocupă cu calcul numeric. Am folosit *Math.NET* pentru calculul matriceal și a normelor  $L^2$ , împreună cu structurile speciale *Matrix* și *Vector* din cadrul librăriei pentru a manipula parametri  $\lambda = \{\mathbf{A}, \mathbf{B}, \pi\}$  ai modelului.

În cadrul librăriei ce descrie modelul Markov cu stări invizibile s-au folosit o mulțime de funcții auxiliare ce ajută la normalizare și structurarea mai bună a codului. Mai jos sunt prezentate cele mai importante funcții auxiliare din cadrul acesteia.

Exemplu 3.2: Funcție auxiliară ce realizează normalizarea

```

1 private Matrix<double> Normalize(Matrix<double> target, Vector<double> divisors, string mode){
2     Matrix<double> output = Matrix<double>.Build.DenseOfMatrix(target);
3     for (int i = 0; i < target.RowCount; ++i){
4         for (int j = 0; j < target.ColumnCount; ++j){
5             if (mode == "row")
6                 output[i, j] /= divisors[i];
7             if (mode == "column")
8                 output[i, j] /= divisors[j];
9         }
10    }
11    return output;
12 }
```

Exemplu 3.3: Funcție auxiliară ce construiește matricea de frecvență  $C$

```

1 private Matrix<double> GetOccurrenceMatrix(List<int> observations){
2     Matrix<double> occurrences = Matrix<double>.Build.Dense(this.emissionCount, this.emissionCount);
3     for (int i = 0; i < observations.Count - 1; ++i){
4         occurrences[observations[i], observations[i + 1]] += 1;
5     }
6     return occurrences;
7 }
```

Exemplu 3.4: Funcție auxiliară ce construiește o matrice cu intrări aleatoare

```

1 private void SetRandomMatrix(double[,] matrix){
2     double[] divisors = new double[matrix.GetLength(0)];
3     double rowMax = 0.0f;
4     for (int i = 0; i < matrix.GetLength(0); ++i){
5         rowMax = 0;
6         for (int j = 0; j < matrix.GetLength(1); ++j){
7             matrix[i, j] = Random.Range(0.0f, 100.0f);
8             rowMax += matrix[i, j];
9         }
10        divisors[i] = rowMax;
11    }
12    Normalize(matrix, divisors);
13 }
```

Legat de aplicarea capitolului teoretic, antrenarea este realizată de o singură funcție căreia îi sunt pasări parametri modelului, împreună cu secvența de etichete observate. Funcția aplică ecuațiile discutate în cadrul capitolului de aspecte teoretice, bineînțele adaptate la limbajul de programare folosit.

### Exemplu 3.5: Implementare algoritmului de antrenare a modelului

```

1 public double Train(List<int> observations, double[,] transitionProbabilities, double[,] emissionProbabilities, List<double> pi){
2     this.SetEmissionMatrix(emissionProbabilities);
3     this.SetTransitionMatrix(transitionProbabilities);
4     this.SetDrawProbabilities(pi);
5     Matrix<double> previousJoinDistribution = Matrix<double>.Build.Dense(this.emissionCount, this.emissionCount);
6     Matrix<double> previousEmission = Matrix<double>.Build.Dense(this.stateCount, this.emissionCount);
7     Matrix<double> previousTransition = Matrix<double>.Build.Dense(this.stateCount, this.stateCount);
8     Matrix<double> jointDistribution;
9     Matrix<double> emissionDelta;
10    Matrix<double> transitionDelta;
11    Matrix<double> occurrenceMatrix = this.GetOccurrenceMatrix(observations);
12    this.SetMatrixBar();
13    this.emissionMatrix = this.emissionMatrix.Transpose();
14    previousEmission = previousEmission.Transpose();
15    double likelihood = 0.0f;
16    for (int i = 0; i < this.maxIterations; ++i){
17        jointDistribution = occurrenceMatrix.PointwiseDivide(this.emissionMatrix.Multiply(this.transitionMatrix).Multiply(this.emissionMatrix.Transpose()));
18        transitionDelta = this.transitionMatrix.PointwiseMultiply(this.emissionMatrix.Transpose()).Multiply(jointDistribution).Multiply(this.emissionMatrix);
19        emissionDelta = this.emissionMatrix.PointwiseMultiply(jointDistribution.Multiply(this.emissionMatrix).Multiply(this.transitionMatrix.Transpose())).Add(
20            jointDistribution.Transpose().Multiply(this.emissionMatrix).Multiply(this.transitionMatrix));
21        this.transitionMatrix = transitionDelta.Divide(transitionDelta.RowSums().Sum());
22        this.emissionMatrix = Normalize(emissionDelta, emissionDelta.ColumnSums(), "column");
23        if (this.transitionMatrix.Subtract(previousTransition).L2Norm() < this.epsilon && this.emissionMatrix.Subtract(previousEmission).L2Norm() < this.epsilon &&
24            jointDistribution.Subtract(previousJoinDistribution).L2Norm() < this.epsilon)
25            break;
26        previousEmission = emissionMatrix;
27        previousTransition = transitionMatrix;
28        previousJoinDistribution = jointDistribution;
29    }
30    likelihood = GetLikelihood(occurrenceMatrix, this.transitionMatrix, this.emissionMatrix.Transpose(), this.emissionMatrix.Multiply(this.transitionMatrix).Multiply(this.
31        emissionMatrix.Transpose()));
32    this.transitionMatrix = Normalize(this.transitionMatrix, this.transitionMatrix.RowSums(), "row");
33    this.emissionMatrix = this.emissionMatrix.Transpose();
34    return likelihood;
35 }
```

Funcția *Train* descrisă mai sus este folosită în clasa mare denumită sugestiv *HMM* pentru a antrena modelul. Modul cum funcționează *Unity* este astfel, toate clasele ce sunt derive din *MonoBehaviour* pot fi atașate pe un obiect din scenă. De aceea există un obiect ascuns în scenă ce are atașat toți generatorii, putând schimba foarte ușor toți parametri modelului chiar din *Inspector*. Din cauza acestui lucru librăria este foarte ușor de preluat și utilizat, aceasta conținând și un modul de salvare/citire a parametrilor modelului.

În continuare se va prezenta o diagramă *UML* ce descrie librăria și relația dintre componente ei. Se poate observa că modelul Markov cu stări invizibile poate lucra complet independent de algoritmul de învățare automată.

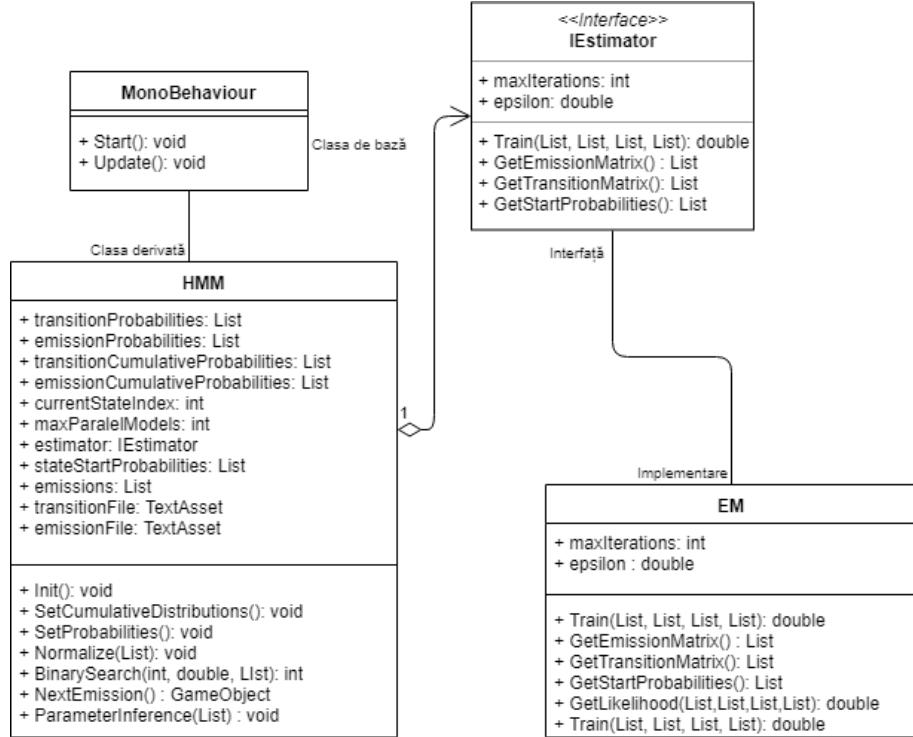


Figura 3.1: Diagrama UML ce modelează libraria de HMM

*IEstimator* este interfață ce specifică ce proprietăți și metode algoritmul de estimare are nevoie să implementeze. Se observă din schemă implementare modelului *Strategy*, ce elimină asocierea dintre clasa *HMM* și algoritmul de antrenare. Din această cauză se poate face foarte ușor schimbarea algoritmului de antrenare în timpul rulării programului în funcție de necesitate.

#### Exemplu 3.6: Folosirea funcție *Train* în cadrul modelului

```

1 public void ParameterInference(List<int> observations)
2 {
3     double[,] bestEmission = this.emissionProbabilities;
4     double[,] bestTransition = this.transitionProbabilities;
5     double[] startProbabilities = this.stateStartProbabilities.ToArray();
6     double maxLoglikelihood = double.MinValue;
7     for (int i = 0; i < this.maxParallelModels; ++i){
8         double likelihood = estimator.Train(observations, this.transitionProbabilities, this.emissionProbabilities, this.stateStartProbabilities);
9         if (maxLoglikelihood < likelihood){
10             maxLoglikelihood = likelihood;
11             bestEmission = estimator.GetEmissionMatrix();
12             bestTransition = estimator.GetTransitionMatrix();
13             startProbabilities = estimator.GetStartProbabilities();
14         }
15         this.SetRandomProbabilities();
16     }
17     this.emissionProbabilities = bestEmission;
18     this.transitionProbabilities = bestTransition;
19     this.stateStartProbabilities = new List<double>(startProbabilities);
20     this.currentStateIndex = this.SetInitialState();
21     this.SetCumulativeDistributions();
22 }
23 }
```

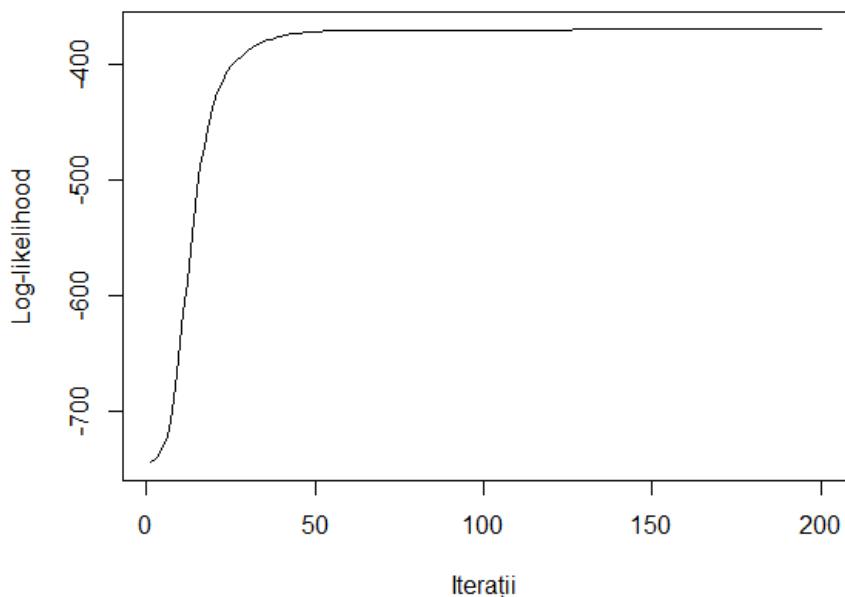


Figura 3.2: Grafic ce prezintă antrenarea unui generator

După cum se observă funcția de *log – likelihood* este monoton crescătoare și modelul de la un anumit număr de interacții converge.

### 3.2 Detalii despre sistemul de instanțiere

Așa cum am mai menționat modelul Markov cu stări invizibile este folosit ca și generator iar pentru instanțiere a fost necesară implementarea unui sistem ce folosește puncte de *spawn* amplasate pe platforme. Odată cu aceste puncte există și un script ce descrie dacă punctul de instanțiere poate să conțină mai multe obiecte sau nu, amplasarea lor facându-se unu după altul. Se poate specifica și rotația obiectului, dar în general se urmărește ca obiectele de pe platformă să fie orientate spre jucător, în sensul acelor de ceasornic.

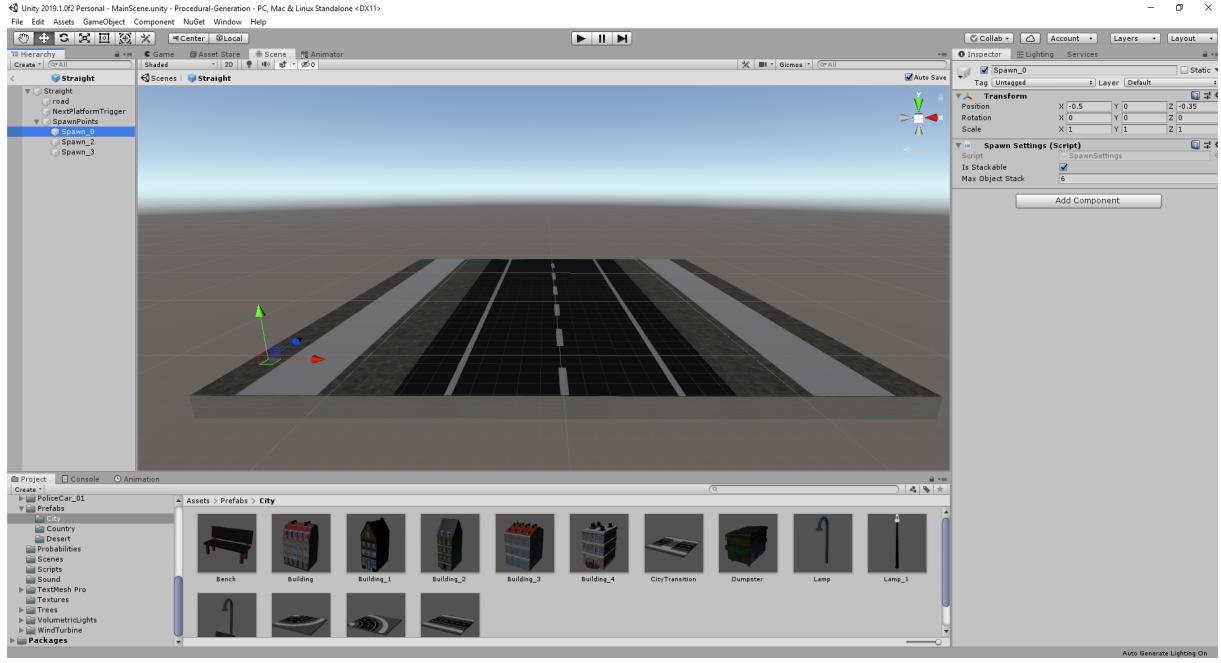


Figura 3.3: Sistemul de instantiere

De asemenea punctele de *spawn* sunt plasate exact pe marginea platformei iar algoritmul de instantiere ia în calcul acest lucru. Cu toate acestea ele pot fi mutate spre interiorul platformei pentru a da senzația de realism. O limitare a acestui sistem este sensibilitatea la numărul de obiecte pe un loc de instantiere, deoarece acesta nu ține cont de lungimea platformei.

### 3.3 Utilizarea librăriei în joc

În dezvoltarea acestei librării obiectivele principale au fost modularitatea și usurința de utilizare. Acest lucru a fost realizat foarte simplu datorită modului în care *Unity* permite atașarea de fișiere sursă pe obiecte din scenă, având la dispoziție toți parametri disponibili din acea clasă.

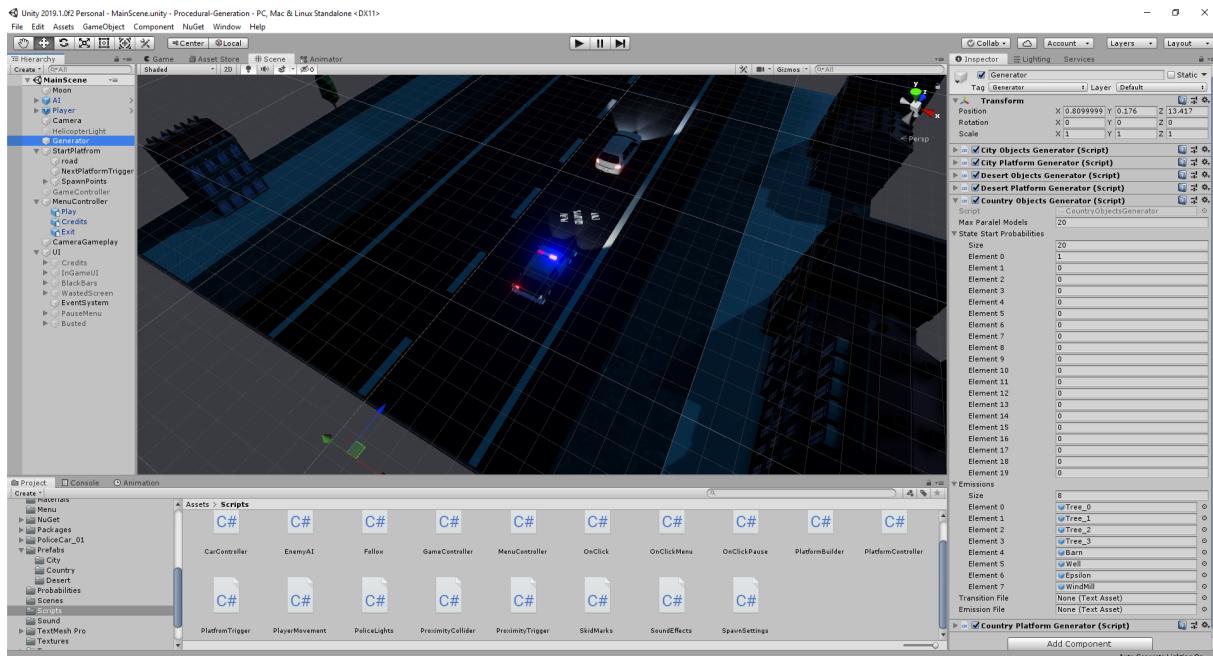


Figura 3.4: Generatorii folosiți în cadrul jocului

În general pentru a putea folosi librăria, se definește o clasă nouă ce moștenește din clasa *HMM*, în cadrul căreia se apelează funcția *Init()* din funcția *Start()*.

#### Exemplu 3.7: Exemplu de folosire a librăriei

```

1 public class DesertObjectsGenerator : HMM{
2     void Start(){
3         this.Init();
4     }
5 }
```

Este necesară această procedură din cauza modului cum funcționează *Unity*, deoarece scripturile nu pot fi atașate pe obiecte în cadrul scenei dacă nu sunt derivate din *MonoBehaviour* iar *C#* nu permite moștenirea multiplă.

Acest lucru face un pic incomodă folosirea librăriei dar capabilitatea de a putea atașa scriptul de orice obiect din scenă și de a putea adăuga referințe direct din *Inspector* este mult prea importantă pentru a nu deriva din clasa *MonoBehaviour*.

Librăria dispune și de o modalitate persistentă de încarcare a parametrilor unui model și anume prin așa numitele obiecte *TextAsset* disponibile în cadrul motorului de joc. De exemplu, după antrenarea modelului acestuia pot fi salvați prin apelarea unei funcții într-un astfel de obiect ce poate fi ulterior referențiat librăriei pentru a putea încărca modelul gata antrenat.

## 3.4 Detalii legate de aspectul aplicației

Un alt aspect ce a necesitat destul de mult timp în cadrul acestei aplicatii a fost partea de *UI*<sup>3</sup> căt și îmbunătățirile grafice aduse prin postprocesare, *SSR*, reflectii în timp real, anti-aliasing, bloom, lumină volumetrică și corectare de culoare.

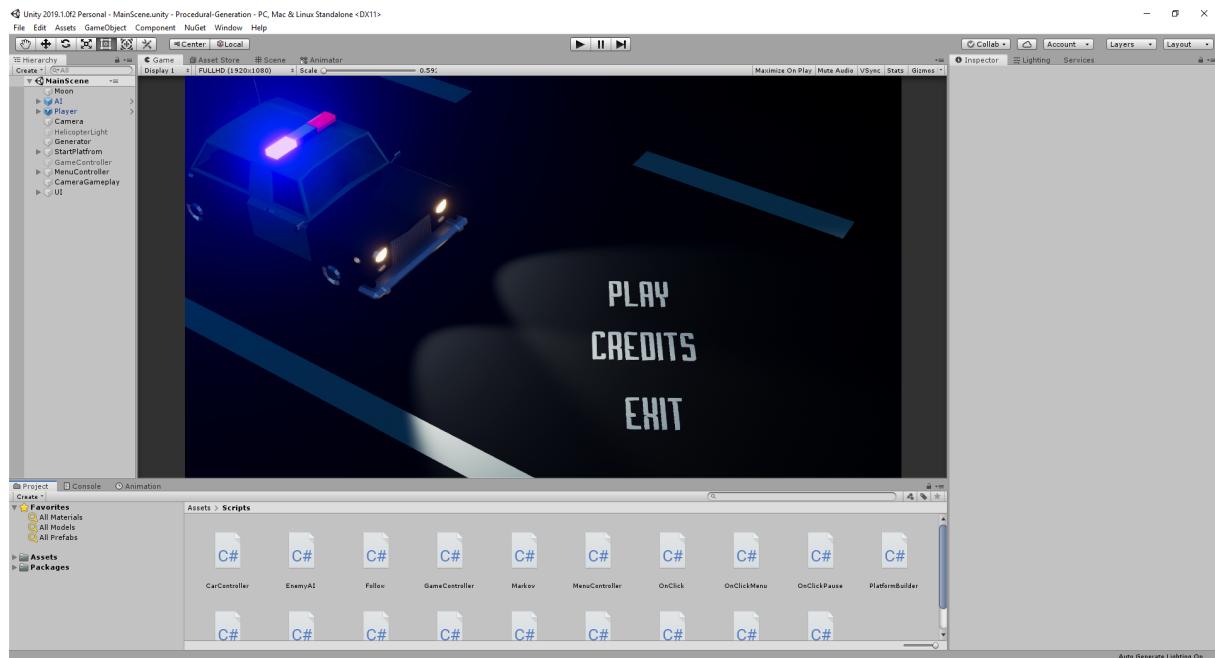


Figura 3.5: Motorul Grafic Unity

Pe lângă partea grafică am încercat să adaug animații atât obiectelor din cadrul scenei căt și a butoanelor cu care interacționează jucătorul. Datorită uneltelor din *Unity* acest lucru a fost ușor de realizat. Pe lângă crearea animațiilor am fost nevoit să utilizez și un automat cu stări finite, unde fiecare nod reprezintă de fapt o altă animație. În *Unity* acestă unealtă se numește *Animator*, și manipulează condițiile de trece de la o stare la altă cât și durata.

<sup>3</sup>User Interface

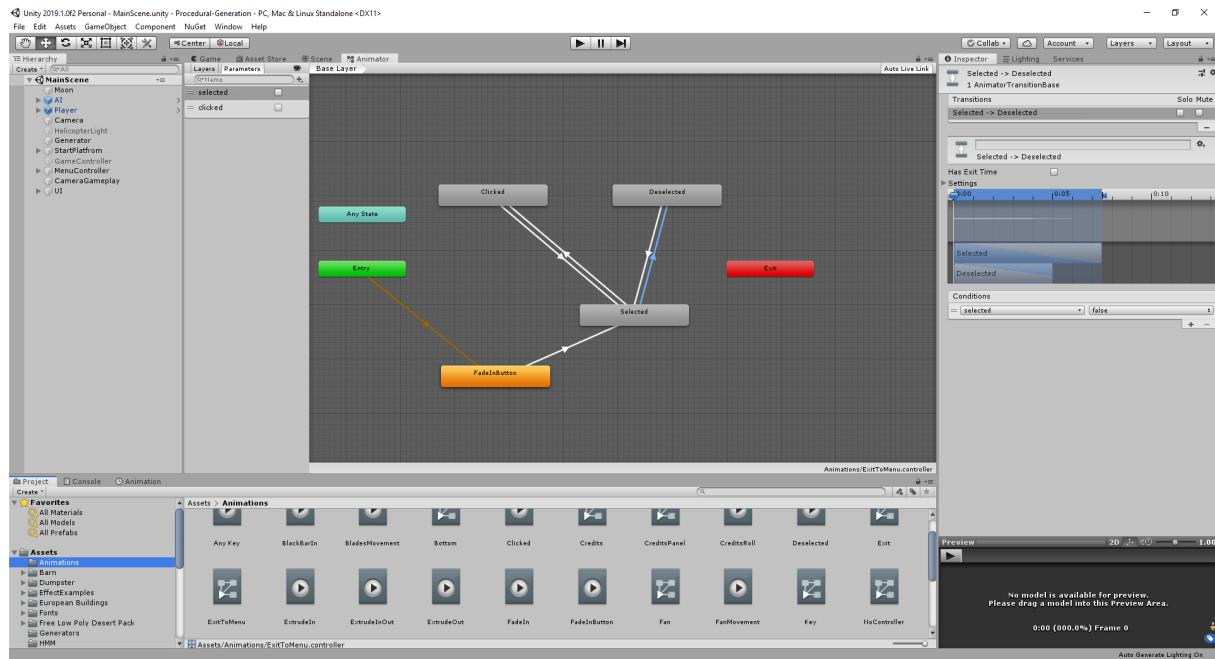


Figura 3.6: Utilitarul folosit pentru animații

Am încercat în mare parte să refolosesc animații și obiecte din motive de eficiență și spațiu de stocare. De accea multe stări din automatele ce se ocupă de animații se repetă. Pentru controlul acțiunii butoanelor din meniuri, comportamentul s-a modelat folosind funcții ce sunt apelate la ieșirea din stare, mai exact la terminarea animației.

#### Exemplu 3.8: Exemplu de folosire a evenimentului *OnStateExit*

```

1  override public void OnStateExit(Animator animator, AnimatorStateInfo stateInfo, int layerIndex)
2  {
3      MenuController controller = GameObject.FindGameObjectWithTag("MenuController").GetComponent<MenuController>();
4      switch (animator.name){
5          case "Play":
6              controller.OnGameStart();
7              break;
8          case "Credits":
9              controller.OnCredits();
10             break;
11         case "Exit":
12             controller.OnGameExit();
13             break;
14         default:
15             break;
16     }
17 }
```

Jocul dispune de un meniu principal, unu de pauză în timpul jocului și unul ce anunță sfârșitul jocului. Mai jos sunt prezentate aceste trei meniuri, împreună cu opțiunile lor.

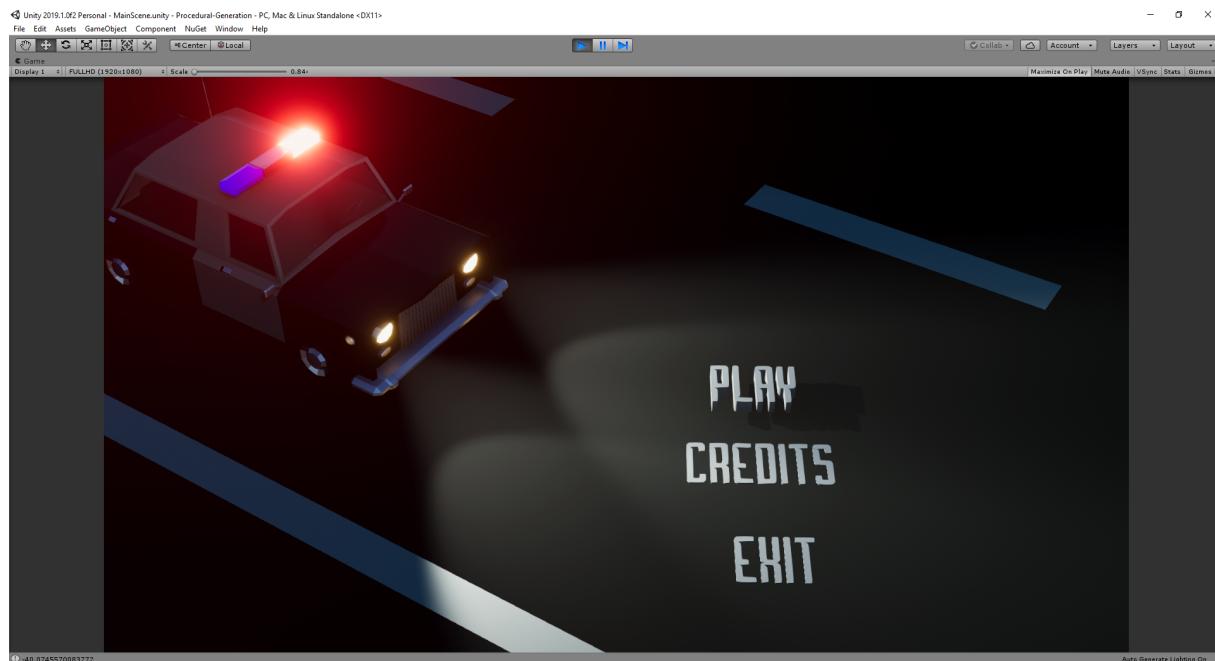


Figura 3.7: Meniul principal din joc

Meniul principal este construit în aceeași scenă ca și jocul principal. Pentru butoanele 3D, ce au fost mai apoi alungite, s-a folosit *Blender* împreună cu fontul *Borg* ce este folosit mai peste tot în cadrul jocului pentru a păstra consistența.

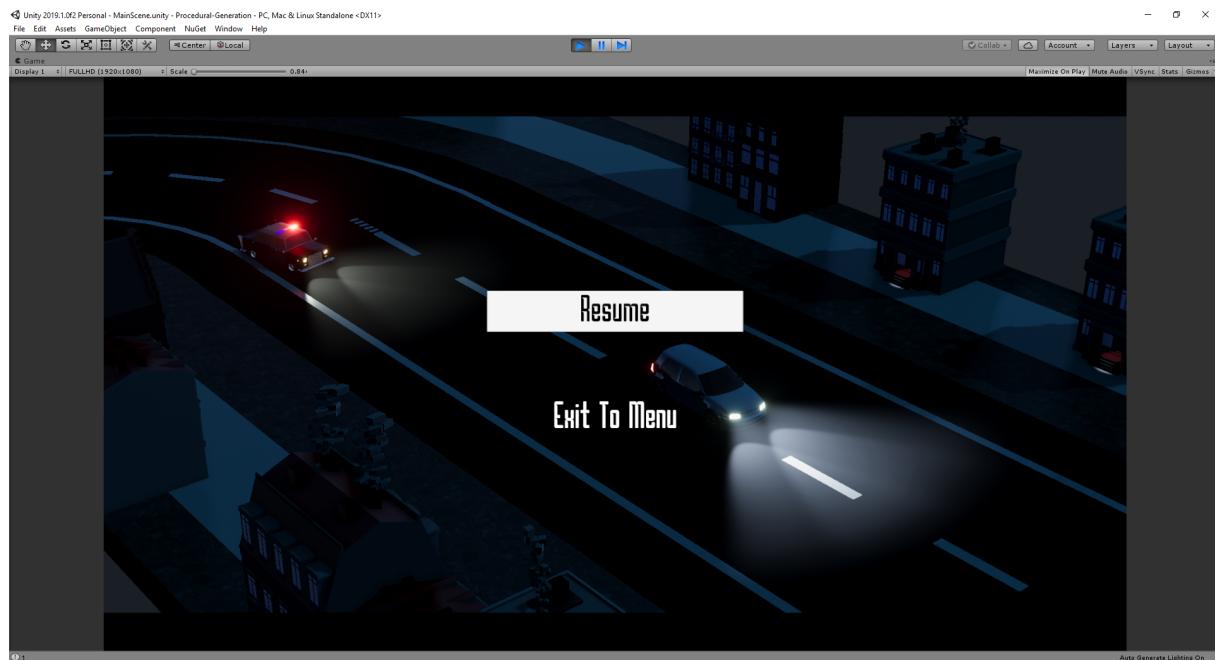


Figura 3.8: Meniul de pauză

Dacă jucătorul este nevoie să oprească jocul acesta poate apăsa tasta *Escape*,

dezvăluind astfel meniul de pauză. De aici acesta poate da *Resume* sau *ExitToMenu* pentru a scoate jocul din starea de pauză, respectiv pentru a reveni la meniul principal. Modul prin care se realizează acest lucru este setarea parametrului *Time.timeScale* la zero respectiv unu.

### Exemplu 3.9: Exemplu de folosire a *Time.timeScale*

```
1 private void ToggleRender() {
2     Time.timeScale = 1.0f - Time.timeScale;
3 }
```

Deoarece jocul este un *EndlessRunner*, condiția de terminare este fie ca jucătorul să cadă de pe platformă fie ca acesta să se afle în proximitatea inamicului ce îl urmărește. Legat de partea de inteligență artificială a adversarului de care trebuie să fugă jucătorul, s-a folosit un simplu script ce urmărește mișcarea acestuia. Bineînțeles această metodă prezintă multe probleme ce pot fi foarte ușor rezolvate cu o abordare ce folosește *Raycasting*<sup>4</sup>.

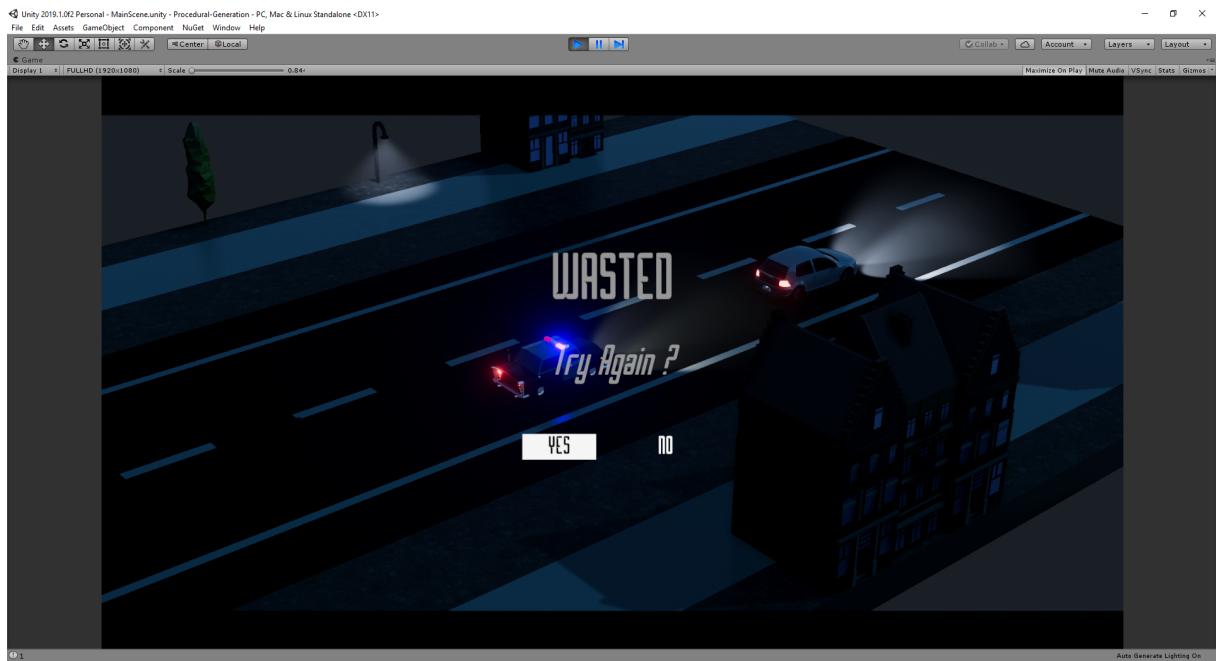


Figura 3.9: Meniul atunci când jocul s-a terminat

În general scopul aplicației a fost validarea abordării de generare cu modelul Markov, de accea unele aspecte ale jocului pot primi îmbunătățiri, mai ales la partea de inteligență artificială discutată mai sus. Am optat să scriu o mică bucată de cod ce

---

<sup>4</sup>Metodă de testare a intersecțiilor dintre obiecte într-o scenă

simulează un agent inteligent foarte primitiv tocmai din această cauză, putând foarte ușor importa o librărie pentru acest lucru.

### 3.5 Detalii legate de controlul mașinilor

Așa cum menționează și titlu pe langă partea de generare și instantiere a fost nevoie de implementarea unui script ce controlează și verifică intrarea de la tastatură pentru a putea controla vehiculul din joc. Pe lângă acest lucru s-au implementat și anumite artificii vizuale ce conferă realism jocului împreună cu utilizarea unei *ReflectionProbe* pentru a simula reflectia în timp real de pe caroseria mașinii. Deși această metodă este costisitoare computațional am încercat să folosesc această tehnică într-un mod căt de optim posibil, cea mai simplă optimizare fiind reducerea dimensiunii texturii generate de aceasta.

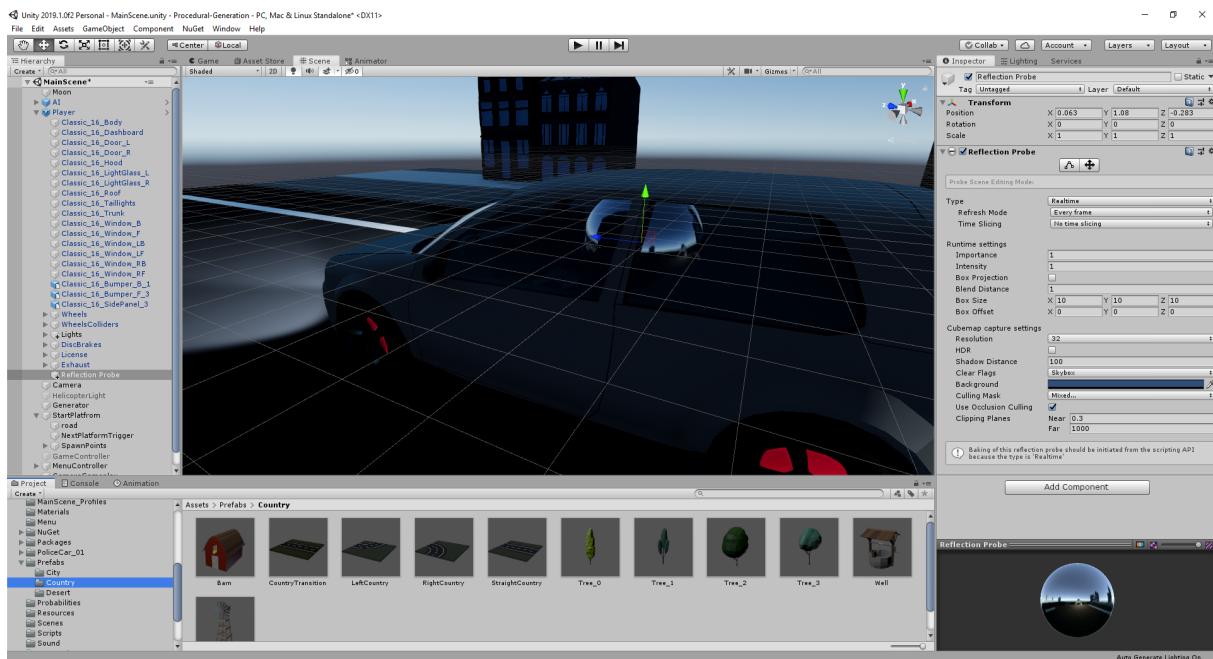


Figura 3.10: Utilizarea componentei *ReflectionProbe* în joc

Pe lângă această unealtă în cadrul mașinilor s-au utilizat și o componentă specială facută pentru interacțiunea dintre o roată și carosabil, aceasta numindu-se *WheelCollider*. Atât jucătorul căt și adversul folosesc această componentă pe fiecare roată, pentru a simula aderență, forță de frânare, acțiunea suspensiei asupra roților și multe altele.



Figura 3.11: Utilizarea componentei *WheelCollider* în joc

## 3.6 Structura întregii aplicații

În general aplicația este construită în jurul uneltelor disponibile din *Unity*, folosind în mare parte sistemul de referințe. Din cauza acestui lucru aplicația este foarte simplă de înțeles și se observă foarte ușor fluxul acesteia după cum se vede din diagrama de mai jos.

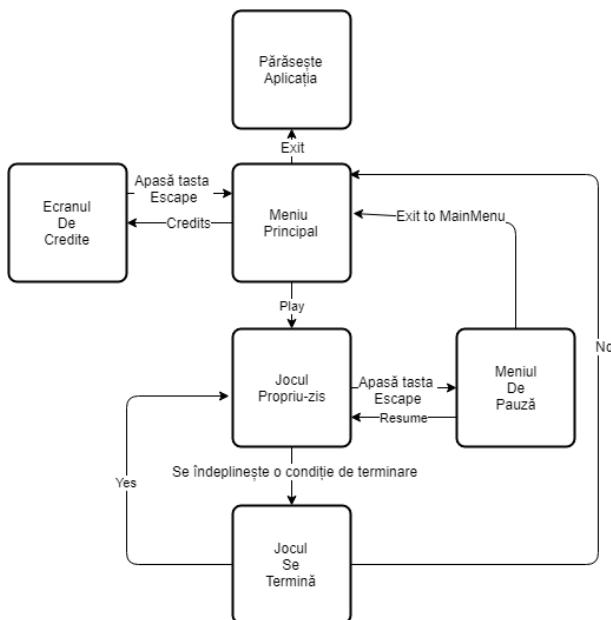


Figura 3.12: Diagramă ce modelează fluxul aplicației

Din punct de vedere a celor două sisteme de generare și instantiere, deoarece sunt decuplate, este necesară o referință a generatorului în cadrul clasei de instantiere. Odată asignată, sistemul de instantiere expune o funcție *InstantiatePlatform* ce este mai apoi apelată când jucatorul declanșează o zonă de pe platformă. De asemenea se face o verificare pentru a nu declanșa instantierea de două sau mai multe ori neintenționat.

Exemplu 3.10: Apelul de instantiere a unei platforme

```

1 void OnTriggerEnter(Collider collidingObject)
2 {
3     if (!this.triggered){
4         this.callbackObject.GetComponent<PlatformBuilder>().InstantiatePlatform();
5         this.triggered = true;
6     }
7 }
```

Mai jos este se poate observa diagrama UML ce modelează comportamentul celor două sisteme. Din necesitatea de a controla numărul de platforme pentru fiecare din cele trei zone, rurală , urbană și desertică s-a introdus un mediator între *PlatformBuilder* și generatori de platforme și anume *PlatformController* ce expune o funcție *GetNextPlatform* ce verifică numărul de platforme de un anumit tip ce au fost generate, înlocuind generatorul, prin rotație, atunci când o anumită limită a fost atinsă, limită ce este determinată aleator după doi parametri ce setează o margine inferioră și superioară.

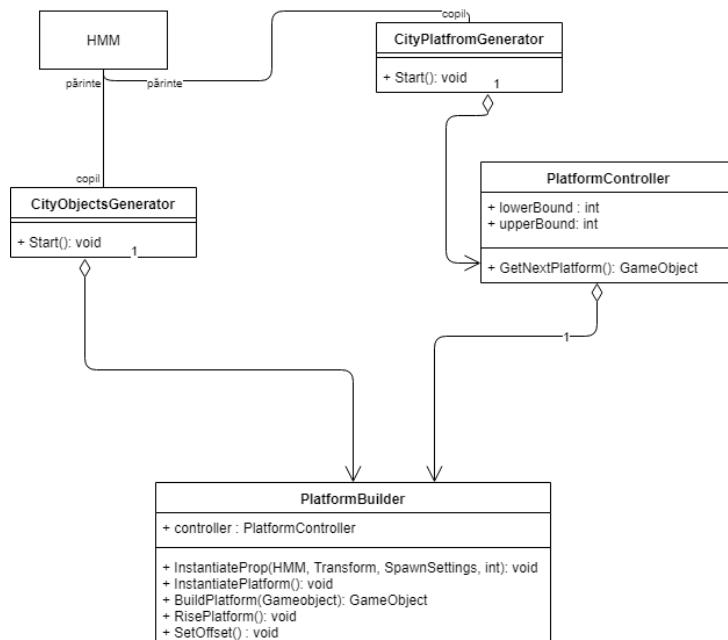


Figura 3.13: Diagramă ce modelează sistemul de generare și instantiere

# Concluzii

Scopul principal al acestei aplicații a fost de a încerca și valida o nouă abordare pentru generare procedurală și de a crea o librărie pentru acest lucru. Din feericire aplicația demonstrează aptitudinile și flexibilitatea acestei abordări împreună cu ușurința de utilizare. Proiectul este *open – source* fiind posibilă utilizarea lui în cadrul oricărei aplicații.

Odată cu terminarea acestei lucrări ce a venit din necesitate, cu siguranță voi utiliza librăria în următoarele proiecte ce utilizează *Unity*, fiind o metodă foarte ușoară de a genera o secvență de obiecte.

Bineînțeles există multe optimizări ce s-ar putea face în cadrul librăriei, primul lucru fiind implementarea metodei lui *Alias* pentru eșantionarea dintr-o distribuție probabilistă discretă, reducând timpul necesar pentru tranzitii și emisii.

Acest lucru ar putea fi luat în considerare doar dacă versiunea curentă ce folosește o căutare binară impune probleme de performanță, pentru aplicația descrisă în această teză fiind suficientă varianta din urmă.

De asemenea cum am mai menționat în cadrul lucrării, agentul intelligent folosit de adversar este unul foarte primitiv, acesta ar putea fi îmbunătățit substantial prin diverse implementări mai mult sau mai puțin complexe. Nu am pus accentul pe acest aspect doarece motivația principală era generarea și instantierea folosind modelul Markov cu stări invizibile.

Legat de platformă pentru care este destinată aplicația, din motive de complexitate al tehnicilor grafice această este disponibilă doar pe desktop, neavând în plan celelalte platforme datorită puterii de procesare reduse. Se poate încerca portare pe alte platforme, dar ar trebui diminuat major din tehnici de consolidare a graficii ce sunt folosite în cadrul platformei actuale.

Cu acest proiect am încercat să demonstreze abilitățile modelului și să încurajeze implicarea mai multor developeri în dezvoltarea acestei tehnici pentru generare.

# Bibliografie

## 3.7 Cărți și Articole

- [1] Madhusudana Shashanka, *A Fast Algorithm For Discrete HMM Training Using Observed Transitions*, 2011,  
<http://cns.bu.edu/~mvss/stuff/ShashankaICASSP2011.pdf>
- [2] Gleidson Mendes Costa, Tiago Bonini Borchartt,  
*Procedural terrain generator for platform games using Markov chain* , 2018, <http://www.sbgames.org/sbgames2018/files/papers/ComputacaoShort/188123.pdf>
- [3] Fanny Yang,Sivaraman Balakrishnan, Martin J. Wainwright,  
*Statistical and Computational Guarantees for the Baum-Welch Algorithm*, 2015,  
<https://arxiv.org/pdf/1512.08269.pdf>
- [4] Sam Snodgrass, Santiago Ontañón,  
*Experiments in Map Generation using Markov Chains*, 2014,  
[http://fdg2014.org/papers/fdg2014\\_paper\\_29.pdf](http://fdg2014.org/papers/fdg2014_paper_29.pdf)
- [5] L. Ciortuz, *Machine Learning Course*,  
<https://profsci.info.uaic.ro/~ciortuz/SLIDES/hmm.pdf>
- [6] *Unity User Manual*,  
<https://docs.unity3d.com/Manual/index.html>

### **3.8 Obiecte preluate și folosite în cadrul aplicației**

- Unity Asset Store, *Low Poly European City Pack*,  
<https://assetstore.unity.com/packages/3d/environments/urban/low-poly-european-city-pack-71042>
- Unity Asset Store, *Low Poly Desert Pack*,  
<https://assetstore.unity.com/packages/3d/environments/free-low-poly-desert-pack-106709>
- Unity Asset Store, *Low Poly Police Car 01*,  
<https://assetstore.unity.com/packages/3d/vehicles/land/low-poly-police-car-01-142826>
- Unity Asset Store, *Low Poly Destructible 2 Cars No.8*,  
<https://assetstore.unity.com/packages/3d/vehicles/land/low-poly-destructible-2-cars-no-8-45368>
- Unity Asset Store, *PBR Dirty Dumpster*,  
<https://assetstore.unity.com/packages/3d/props/exterior/pbr-dirty-dumpster-59840>
- Unity Asset Store, *Unity Particle Pack 5.x*,  
<https://assetstore.unity.com/packages/essentials/asset-packs/unity-particle-pack-5-x-73777>
- Unity Asset Store, *Post Processing Stack*,  
<https://assetstore.unity.com/packages/essentials/post-processing-stack-83912>
- Joao Paulo, *Low Poly Farm*,  
<https://3dshed.wordpress.com/2018/08/29/low-poly-farm-animation/>
- Joao Paulo, *Wind Turbine*,  
<https://3dshed.wordpress.com/2018/02/12/wind-turbine-001-animated/>
- Joao Paulo, *Trees Pack*,  
<https://3dshed.wordpress.com/2018/04/07/trees-pack-001/>

- Joao Paulo, *Desert Texture*,  
<https://3dtextures.me/2018/08/13/cracked-mud-001/>
- Joao Paulo, *Asphalt Texture*,  
<https://3dtextures.me/2018/05/07/asphalt-005/>
- Joao Paulo, *Stone Tile Texture*,  
<https://3dtextures.me/2018/09/27/stone-tiles-003/>
- Joao Paulo, *Grass Texture*,  
<https://3dtextures.me/2018/01/05/grass-001-2/>
- SlightlyMad, *Volumetric Lights*,  
<https://github.com/SlightlyMad/VolumetricLights>
- DavidMenke, *Main Menu Sound*,  
<https://freesound.org/people/davidmenke/sounds/319750/>
- Kenney.nl, *Buttons Effects*,  
<https://opengameart.org/content/51-ui-sound-effects-buttons-switches-and-clicks>
- Borg Font,  
<https://www.dafont.com/borg.font>